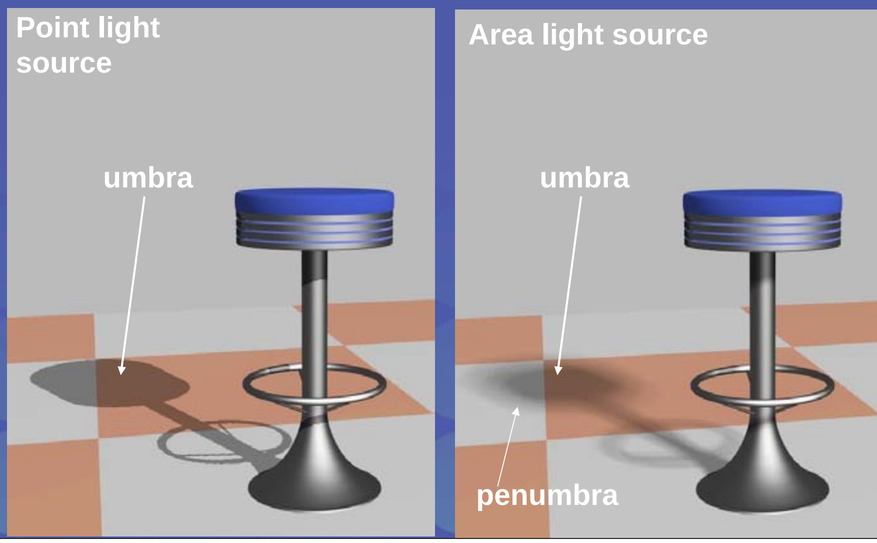
A Soft Shadow Volume Algorithm

- Contents of my presentation
  - Motivation
  - Penumbra wedges
  - Visibility calculations
  - Implementations (SW/HW)
  - Load balancing
  - Disadvantages
  - Comparison: Hard vs Soft
  - Live demo

Q1: Why soft?

- Answer #1:
  - Soft shadows are addictive!



Reasons for soft shadows:

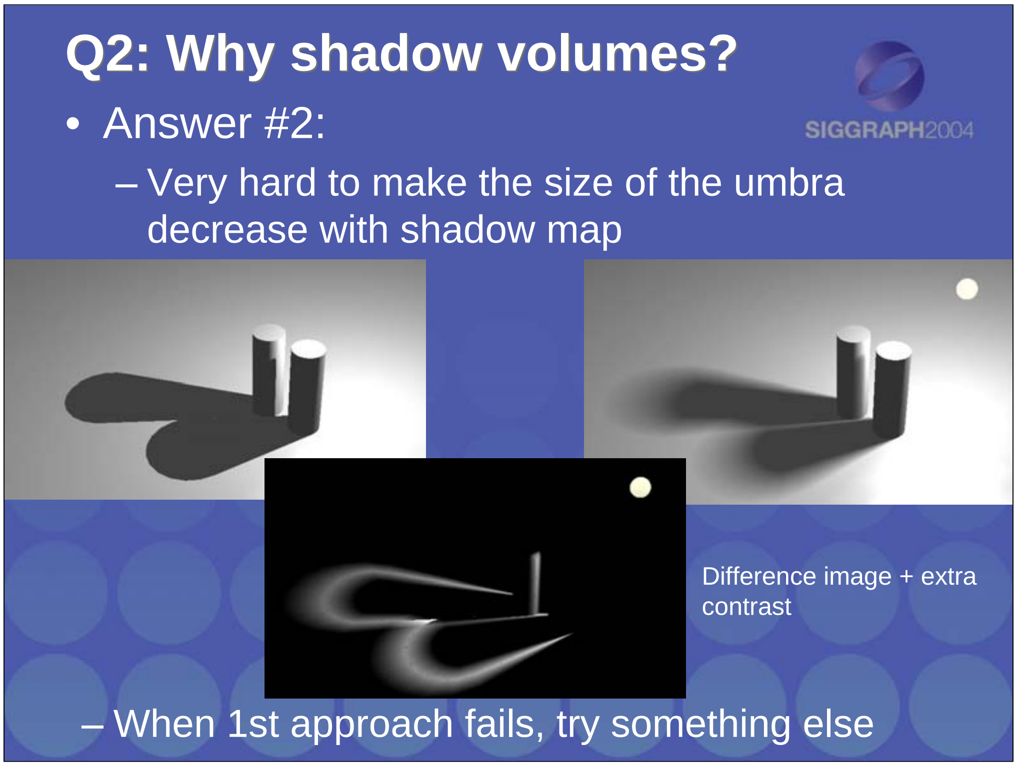
1) increases the level of realism of the rendered images – the large marjority of all light sources have some extensions in space (even the sun)

2) spatial relationsships get even simpler to determine for a human, since sharp shadow edges imply that the shadow caster is close to the receiver, and vice versa.

3) puts off the focus from the shadows, i.e., a hard shadow can sometimes be misinterpreted for a geometrical edge, but that is hardly ever the case with soft shadows

4) Atmosphere: imagine a setting sun...

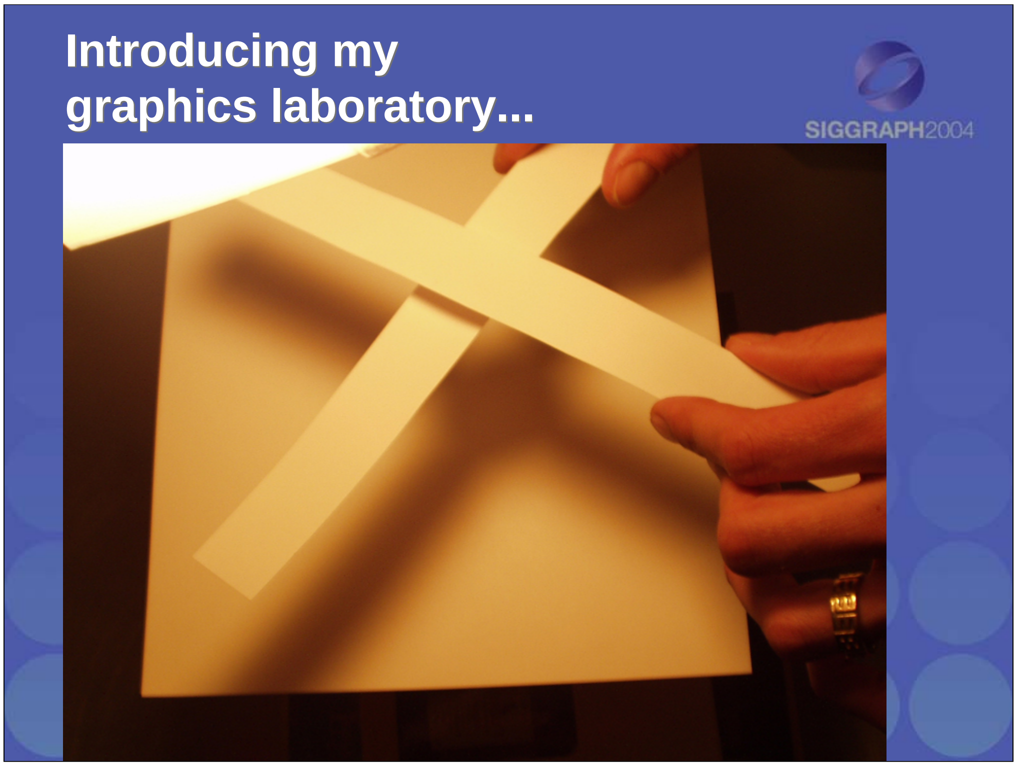
For animated real-time graphics, the addiction is even more severe...



Reasons for shadow volumes:

1) Shadow maps seemed harder to extend into handling soft shadows, that is, the size of the umbra must decrease when light source size increases

2) The complexity of shadow volumes are often considered to be much higher, but in research one should not really put any limits on what to do research on (if the results are convincing enough, but too slow, then let's do research on accelerating the shadow rendering)



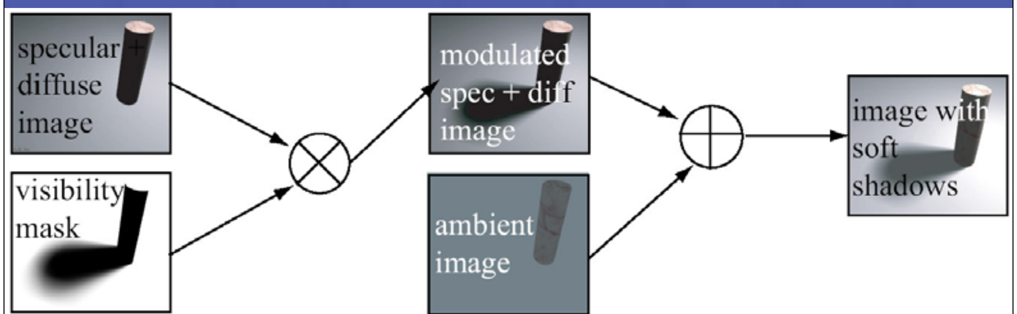
Do this in the kitchen, and you'll learn stuff that is hard to learn at other places!

Cut a square hole in a hard paper, and place a thin paper over the hole. This thin paper should spread the light diffusely. Place the hard + thin paper construction over the light source. Turn off all other light sources, then play with various shadow casters and receivers.

Soft shadow algorithms in general

SIGGRAPH2004

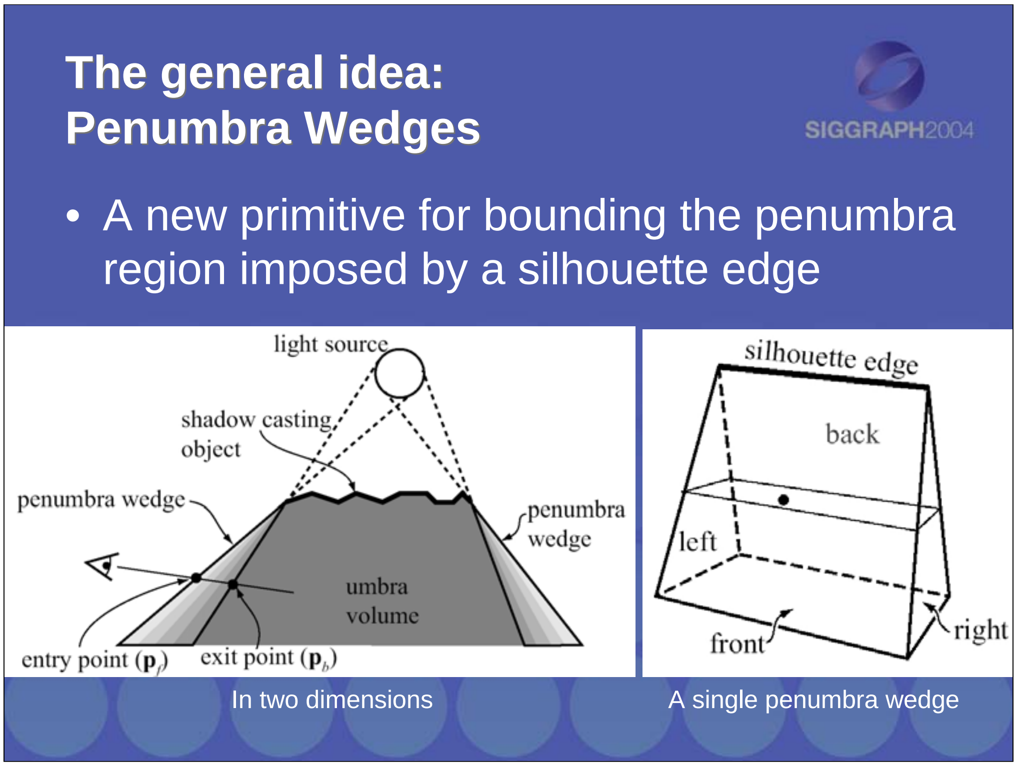
- Fundamental visibility problem in CG
  - Soft shadow rendering and view cell occlusion culling are essentially the same thing
  - Inherently difficult



Lots of research on this every year

Often at least one SIGGRAPH paper on the topic

Difficult because need to have visibility information for each point to be shaded to every point on the area light source



This is the basic INSIGHT behind our algorithm:

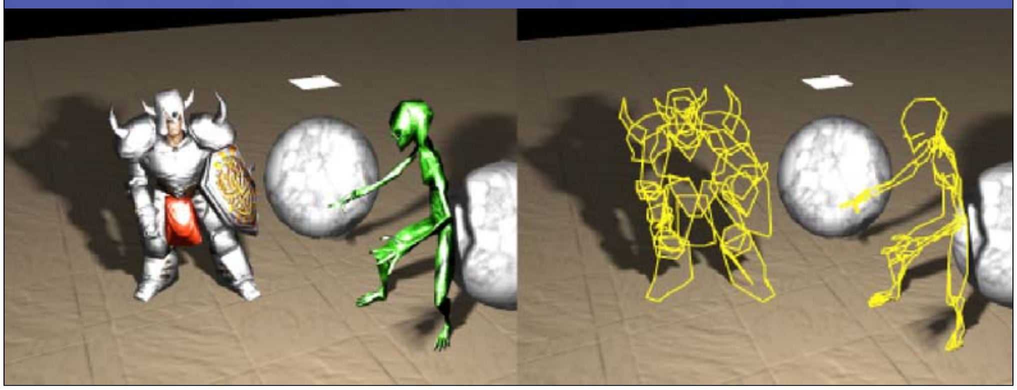
Penumbra wedges are part of the core of the algorithm, without the penumbra wedge, we will have a hard time implementing this algorithm.

The nice thing about the penumbra wedge is that it is possible to rasterize them quite efficiently using today's graphics hardware. This is true, even though the wedge is a 3D entity (not 2D, like a triangle or quad).

Can use more planes than just 5 if you want to.

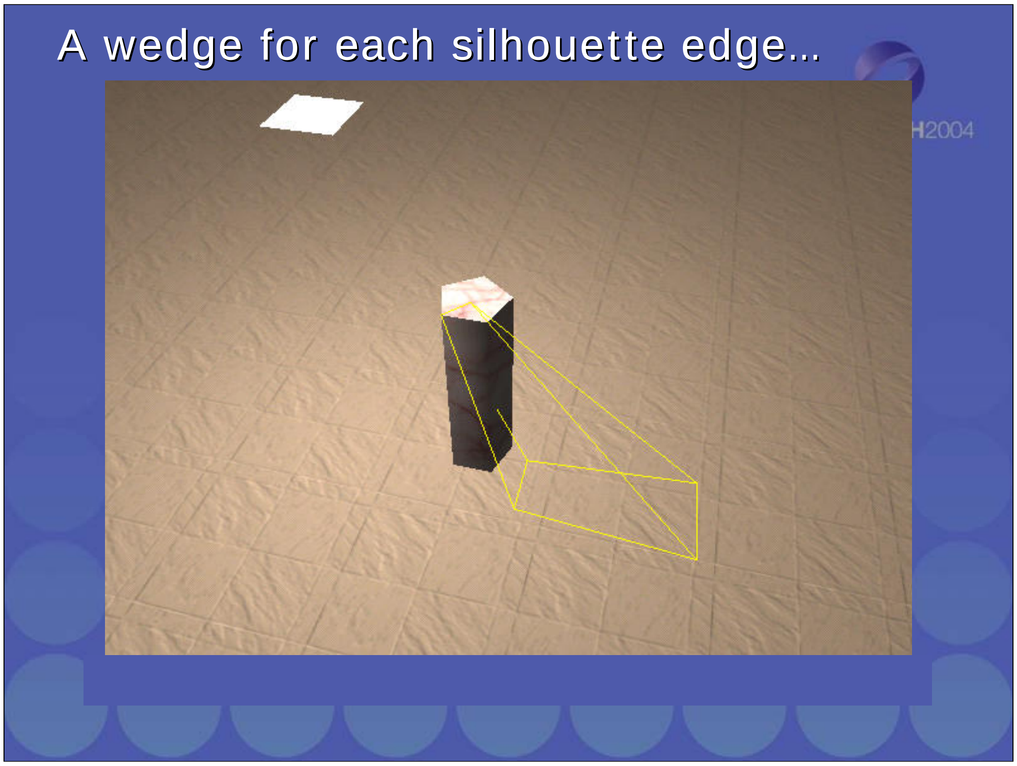
Important simplification

- Compute potential silhoutte edges as seen from the center of the light source
- Use one penumbra wedge per such edge

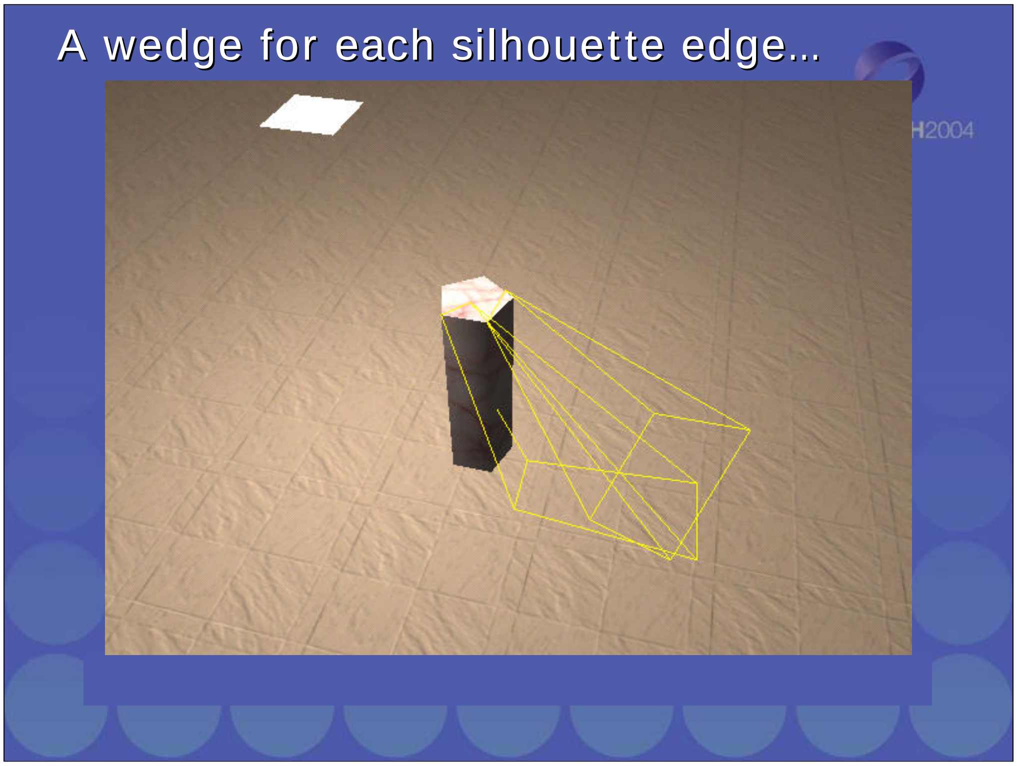


This simplification makes the problem much simpler to solve.

Disadvantage: popping can occur for simple objects, such as a cube.



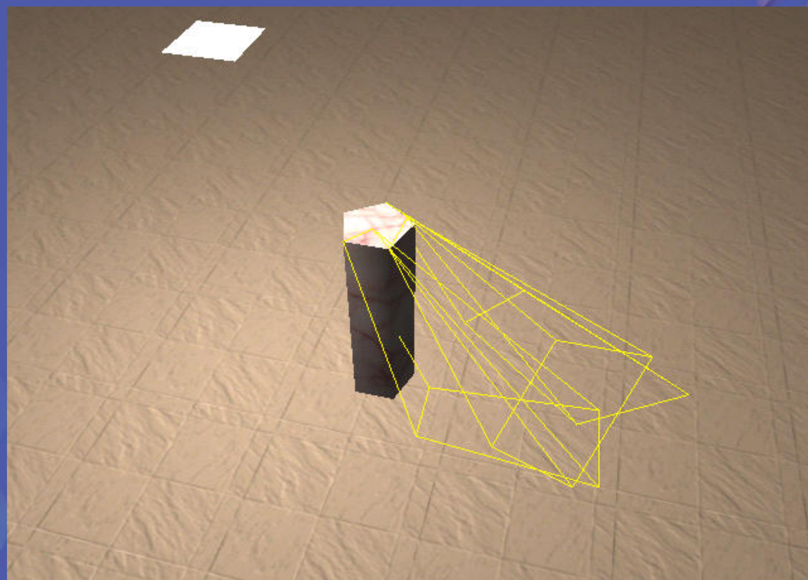
A wedge is generated for each silhouette edge, enclosing a part of the penumbra region .

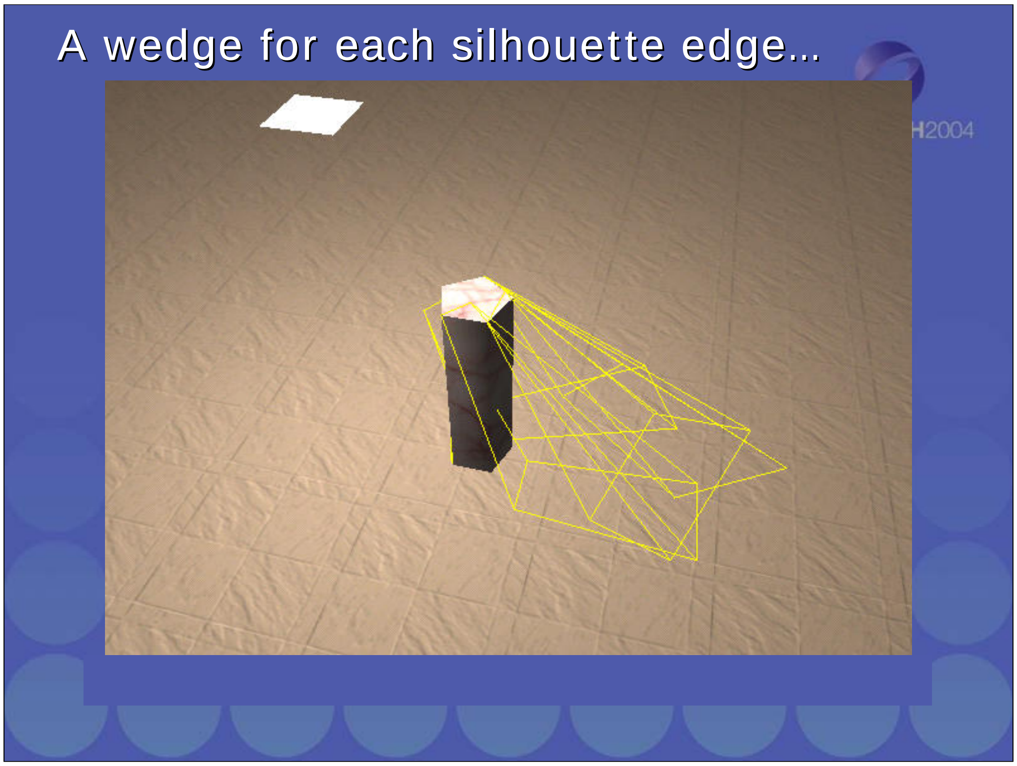


Together, the wedges will enclose the whole penumbra region.

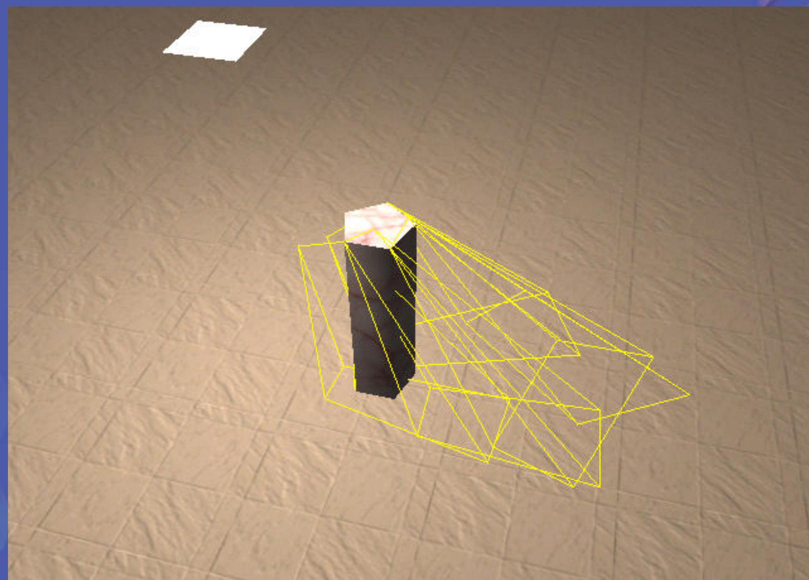
They don't have to correspond exactly to the penumbra region – it is sufficient that they enclose it. And this is a major advantage of our algorithm, since the correct penumbra region can be complicated to compute.

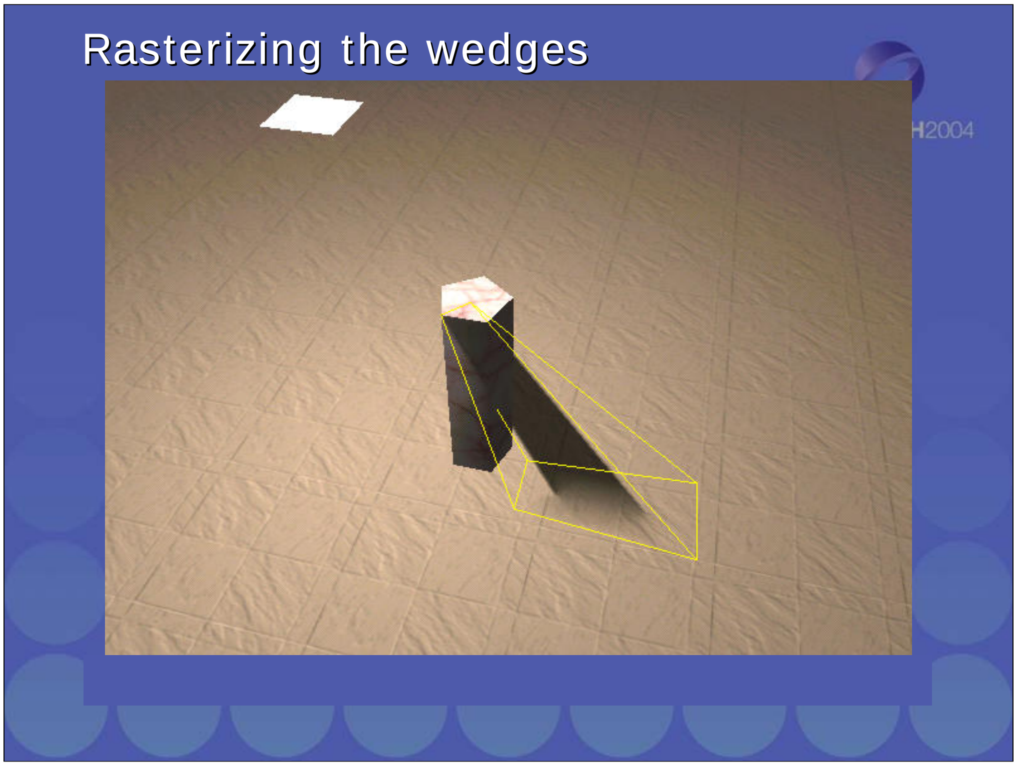
A wedge for each silhouette edge...




A wedge for each silhouette edge...
H2004

A wedge for each silhouette edge...





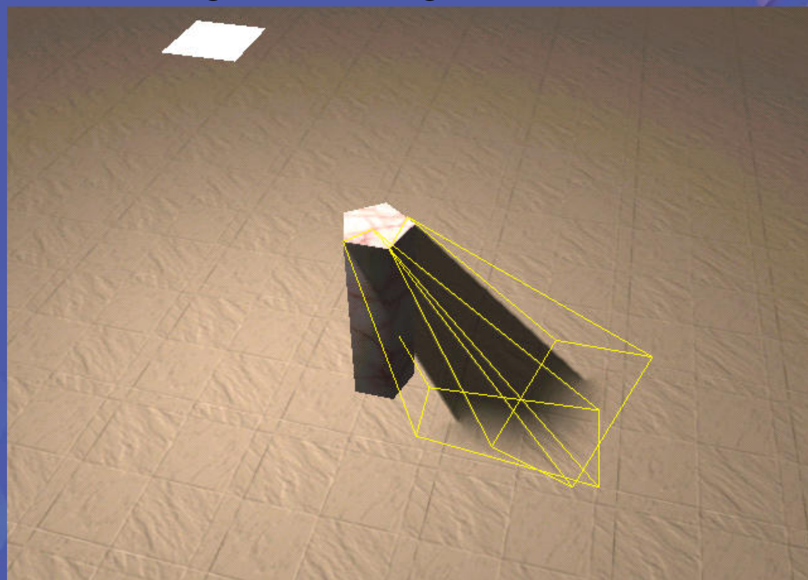
We then rasterize each wedge with a pixel shader.

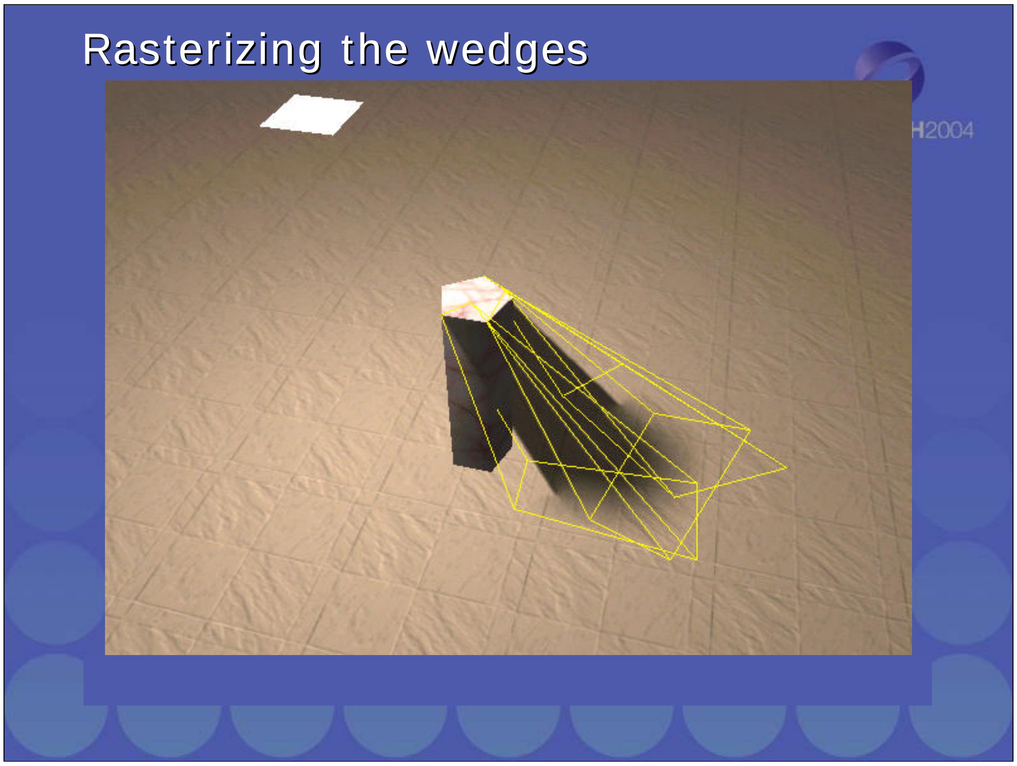
The scene is first rendered into the frame buffer and z-buffer.

The umbra and penumbra contribution is then rasterized, wedge by wedge by our algorithm. The pixel shader reads out a point from the z-buffer and uses that point to compute a shadow contribution that is stored in a separate buffer. And this buffer is later used to modify the whole frame buffer to ”add” on the soft shadows to the image.

I have here visualized the rendering of the umbra contribution and penumbra contribution simultaneously. Typically, we use Crow’s shadow volume algorithm for hard shadows first to fill the umbra, and then compensate with our penumbra pass.

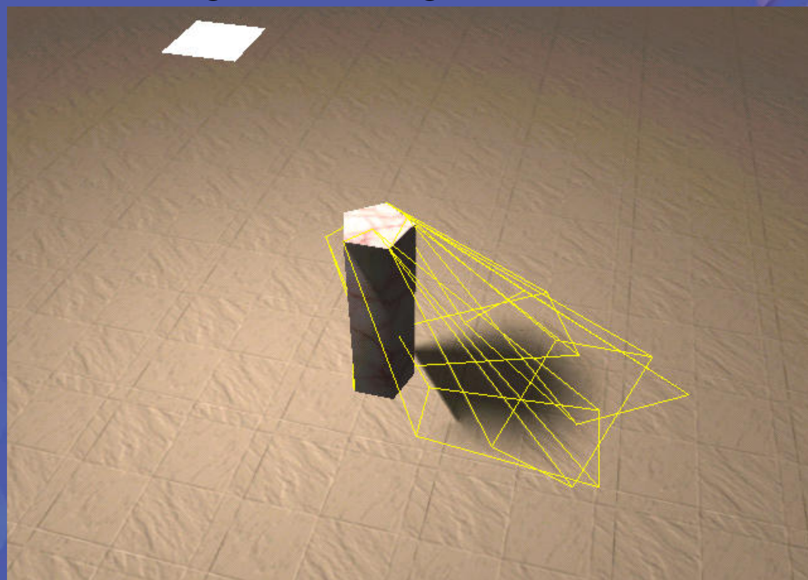
Rasterizing the wedges



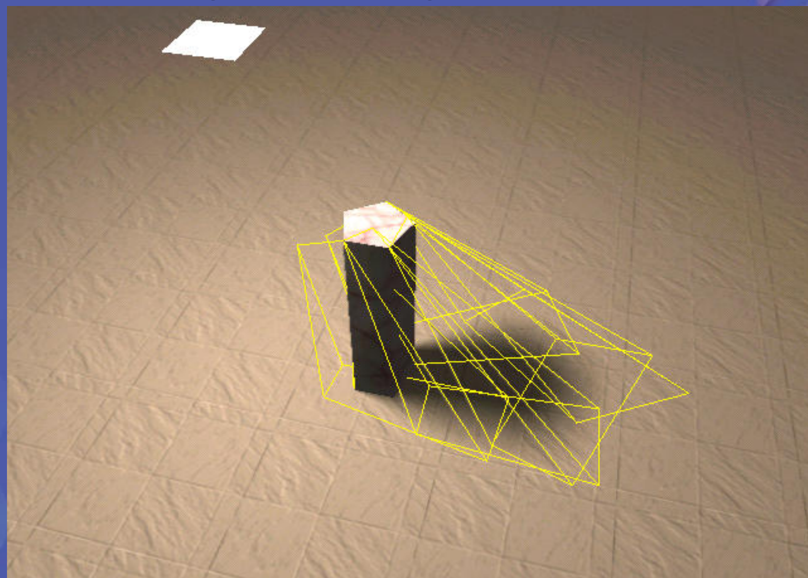


And as they are rasterized, the final soft shadow will gradually appear.

Rasterizing the wedges



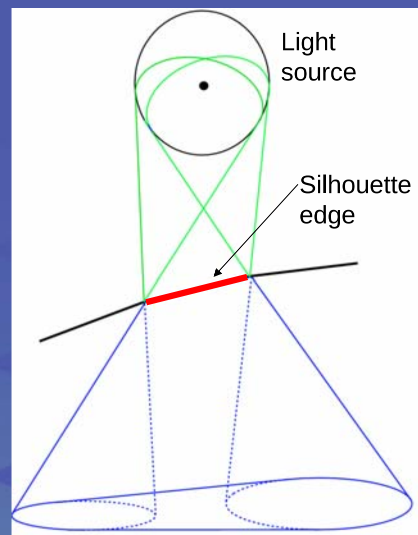
Rasterizing the wedges



Wedge construction: in theory

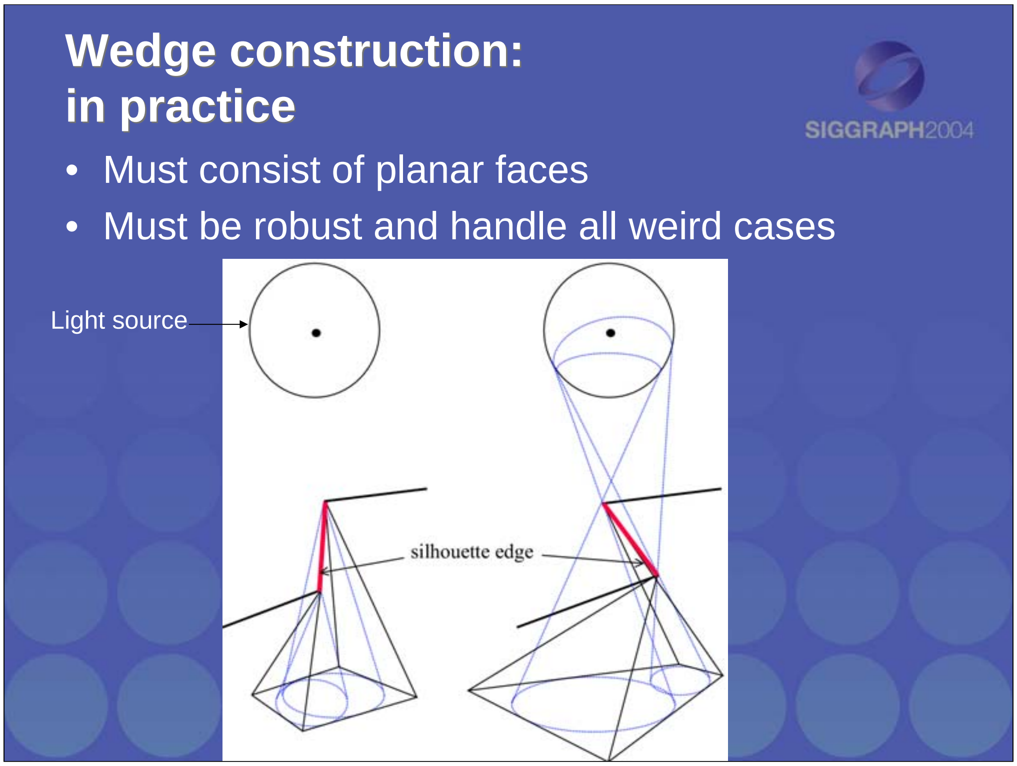
1. Form a cone from light source to edge vertex
2. Sweep the vertex from one end to the other
3. The swept surface is the ”best” wedge

- Not planar: consists of Coon’s patches



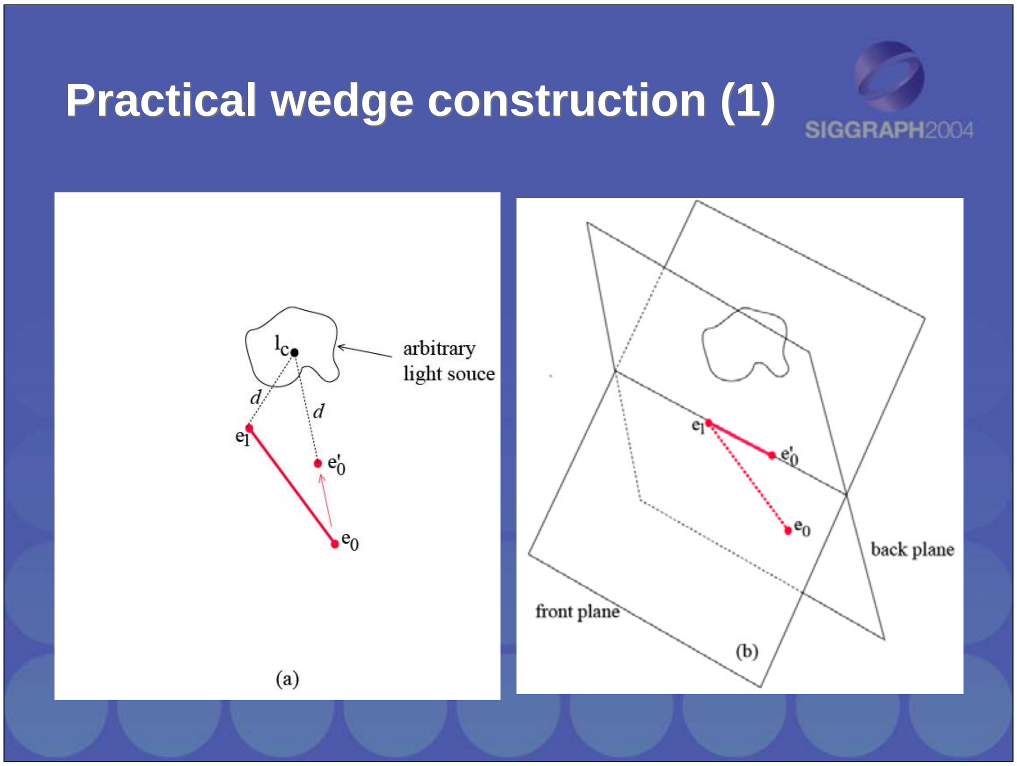
SIGGRAPH2004

Can do exactly this with any light source.



Must be planar because we need to rasterize them!

These cases are difficult because either one cone intersects the other edge end point or cones intersect with each other or both.



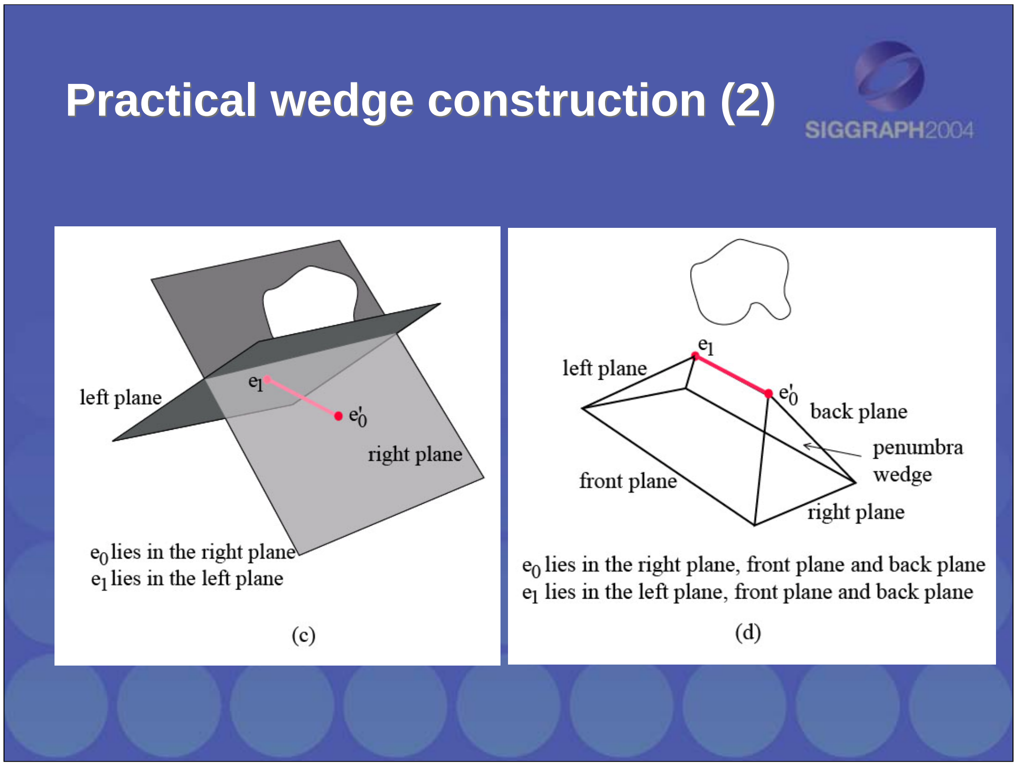
A: first move the farthest edge end point (in this case e0) towards the center of the light source, and stop when the new point is as far from l_c as the other edge end point e1. This makes it possible to get planar faces on the wedge.

B: Place two planes that passes through the edge e0’->e1.

Rotate the first plane until it barely intersects with one far side of the light source.

Rotate the other plane until it barely intersect with the other side of the light source.

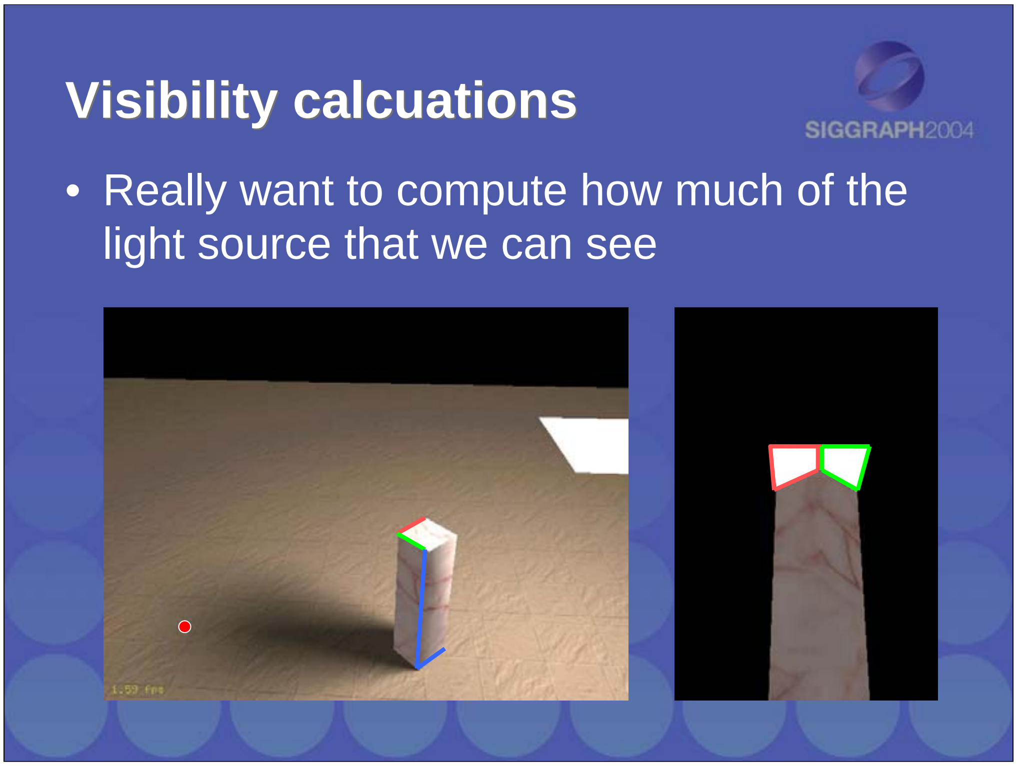
At this point we have created two of the planes of the wedge, namely, the front and back plane.



C: Create left and right planes.

Example: for the right plane, create a plane that can rotate a around an axis that goes through e0´ and passes through the vector which is formed as the cross product of e1->e0´ and l_c->e0´ (where l_c is the center of the light). Then rotate this plane until it barely touches the far side of the light. Do similar things to create the left plane.

D: at this point all plane have been created, and we might limit the extension of the wedge by placing a bottom plane for the wedge as well.



In the left image, the silhouette edges as seen from the center of the light source are marked with blue, green, and red lines.

Imagine that you can jump down to the red dot, and look up towards the light source. Then you see the image to the right.

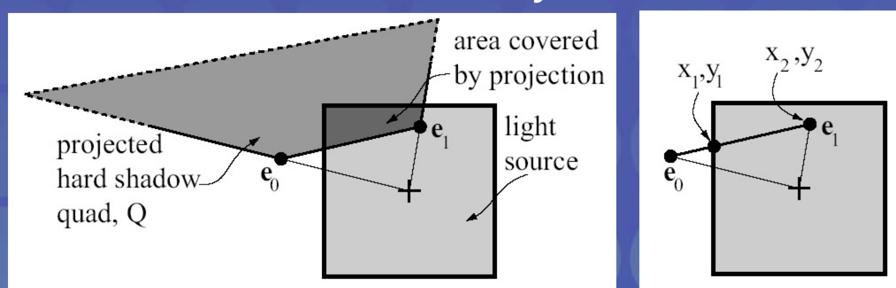
The only silhouettes that project onto the light as seen from the red dot is the red and green silhoutte edges.

They each compute a contribution of how much they can ”see” of the light source.

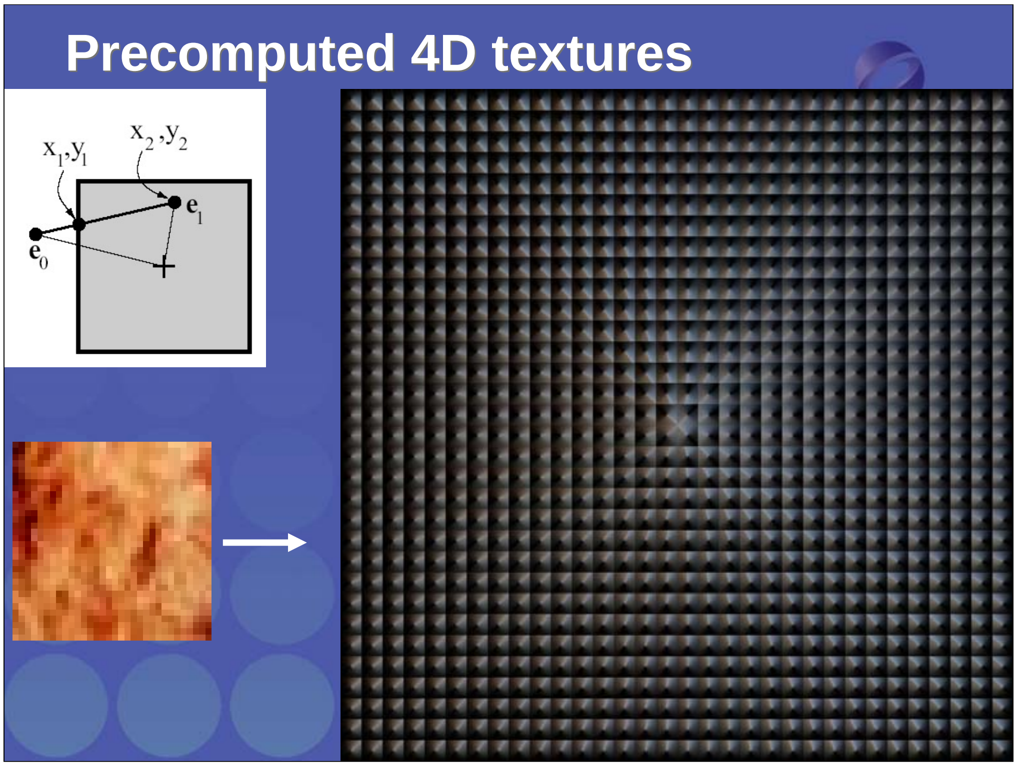
Each silhouette edge's contribution to visibility

SIGGRAPH2004

- From the point-to-be-shaded, project the edge onto the light source
- Compute the area, called *coverage*, of the dark gray region
- Add/subtract to a visibility buffer



The visibility buffer will be described in detail later



Next we clip the projected edge against the borders of the light source (in this case a square light).

This gives us four coordinates: x1,y1, x2,y2.

And thus we can precompute the coverage based on these coordinates into a 4D texxture (which when flattened out

Looks like the image to the right).

In fact, this even allows us use colored textures as ligth sources. Just precompute the sum of the colors of the coverage area (dark gray in previous slide).

Why 4D textures?

- Used as a look-up table
- Due to intelligent caches → fast!
- Can have textured lights (and animated)
- 32x32 light texture → 3 MB

Higher resolution would give better quality, but seldom a problem in practice.

Left: this image was rendered using a single light source, but the texture on it shows 4x4 small area light sources. Due to the precomptued 4D texture, this can be handled in one pass.

Right: a simpler case that shows that we're doing the right thing.

A soft shadow volume algorithm

- 1st pass: Render hard shadow quads (as usual)
  - To be sure that we register when we enter/exit umbra
- 2nd pass: compensate for overstated umbra

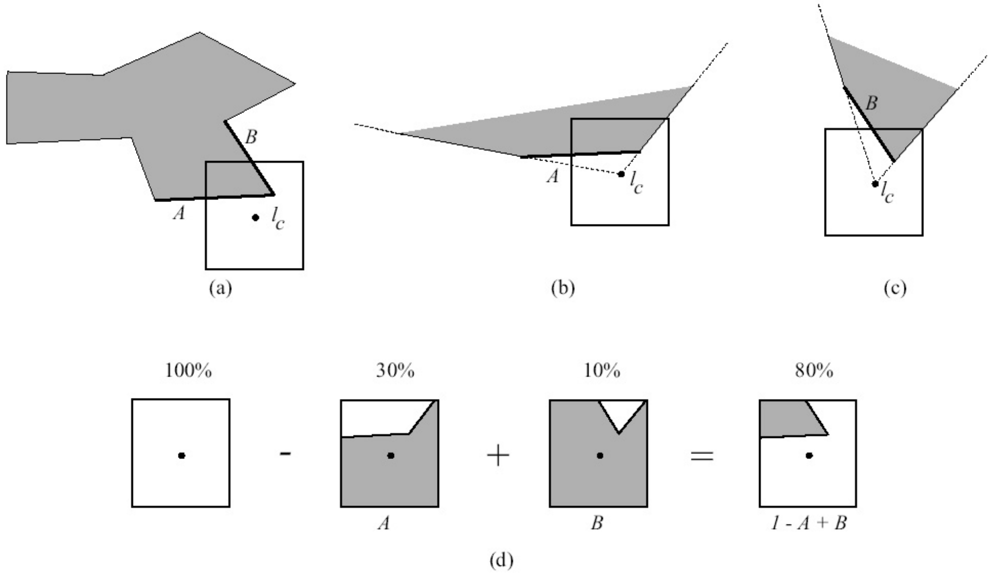
```
rasterizeWedge(wedge W, hard shadow quad Q, light L)
for each pixel (x,y) covered by front facing triangles of wedge
   p = point(x,y,z);   // z is depth buffer value
   if p is inside the wedge
      v_p = projectQuadAndComputeCoverage(W,p,Q);
      if p is in positive half space of Q
         v̄(x,y) = v̄(x,y) − v_p;   // update V−buffer
      else
         v̄(x,y) = v̄(x,y) + v_p;   // update V−buffer
      end;
   end;
end;
```

The coverrage can be at most 0.5, and the hard pass adds 1.0 when whe enter the hard shadows.

Therefore, when the point is in the positive half space of Q, we need to subtract the coverage, and otherwise add it.

Example of how it works...

SIGGRAPH2004



A: the square is a light source with center l_c, and the gray polygon is a shadow casting object. Imagine that we're at the point to be shaded a look up against the shadow caster (the gray object) and the light source.

The only projected edges that can influence visibility is *A* and *B*, and their respective contributions are shown in figure B and C.

D: At the bottom we show the light source as fully visible at first (leftmost image), and then subtract the contribution of *A* due to its orientation with respect to l_c, and then because the orientation is reversed for *B*, we add its coverage value. The result is the expected visibility of the light source.

SW vs HW implementation

SIGGRAPH2004

- Implemented everything in SW due to lack of HW
- Performance was poor when we finally got a higly programmable graphics card
  - *fps*(SW rendering) > *fps*(HW rendering)!
- Had to fine tune the implementation:
  - Tighter wedges for rectangular lights
  - Optimized pixel shaders
  - Frame buffer blending
  - Culling

We will not cover the ”tighter wedges” – see our Graphics Hardware 03 paper.

Implementation – overview

- Compute position buffer (once per frame)
- Compute hard shadows into V-buffer
  - Give overestimation of umbra
- Correct for that by rendering the wedges
  - Use culling for faster rendering
  - Split into 2 buffers: one additive, one subtractive
- Additive – subtractive = shadow mask
- Combine shadow mask with rendered scene

There are several limitations on current graphcis hardware that makes the implementation a bit ackward.

This outline shows how we currently do it, but that can change with a newer graphics card.

Position buffer

- The subsequent fragment programs need the (x,y,z) of each pixel
- Interpolate world space coordinates over each triangle (using texture coordinates)
- One vertex & fragment program compute the (x,y,z) and stores in a floating point pixel buffer (R,G,B)=(x,y,z)

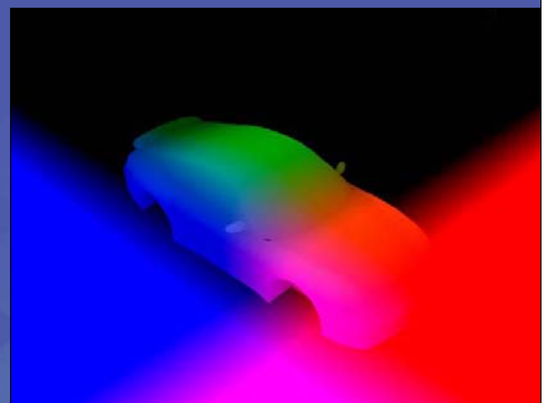
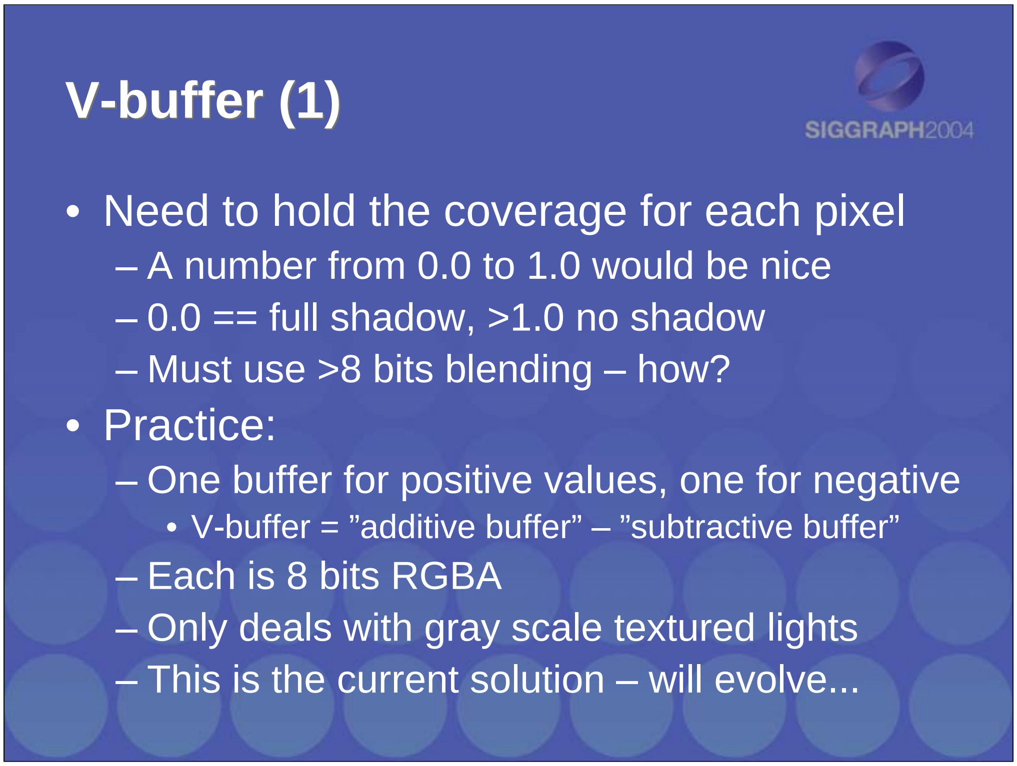


Image courtesy of Jonas Svensson and Ulf Borgenstam

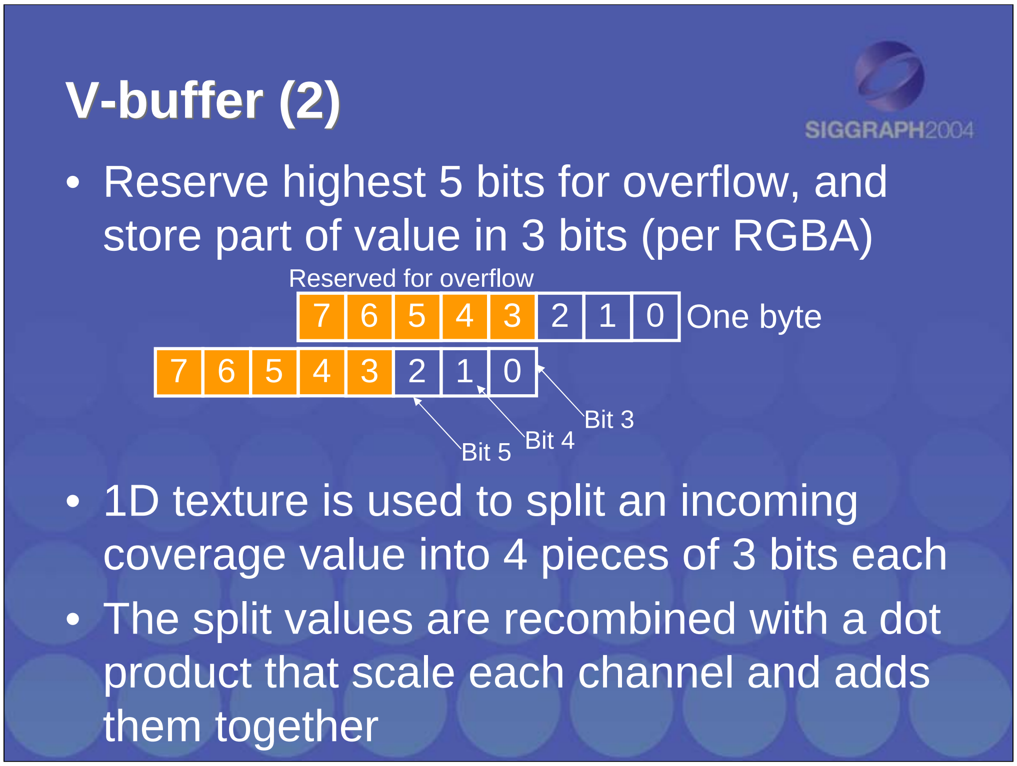
The position buffer is computed once at the beginning of each frame



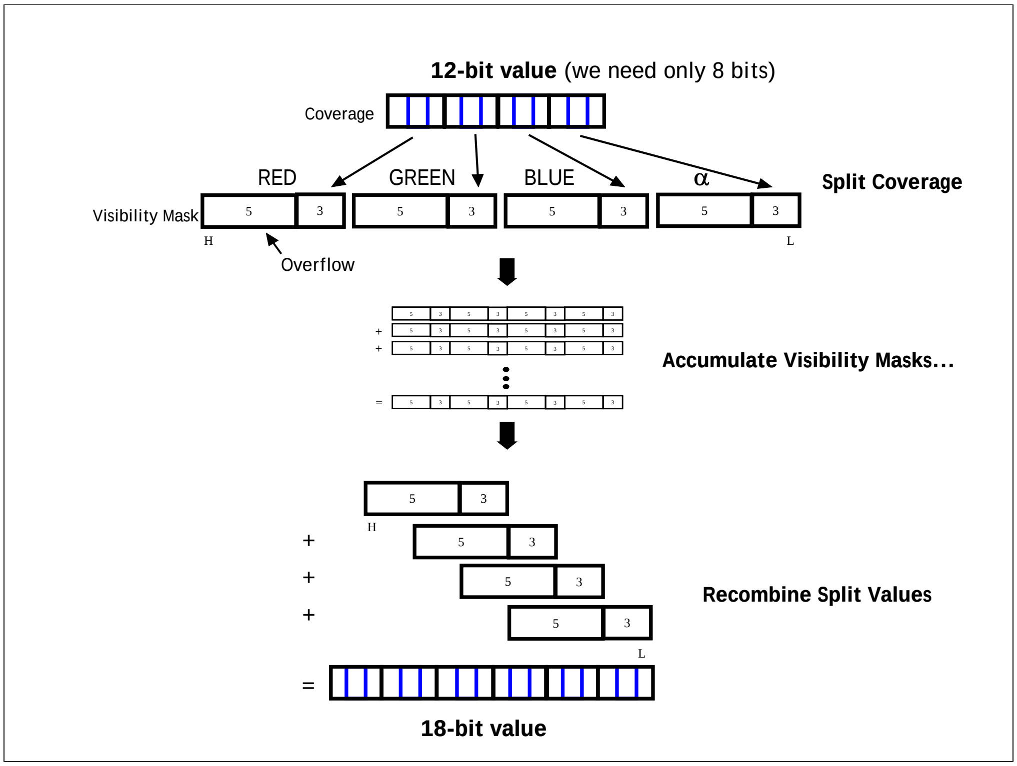
Normal stencil shadows use 8 bits of stencil per pixel. This is so we can have several overlapping shadow volumes.

Here each coverage value should be somewhere between 0 and 255, and we would need

Overlapping objects as well. This we need about 12-16 bits per pixel.



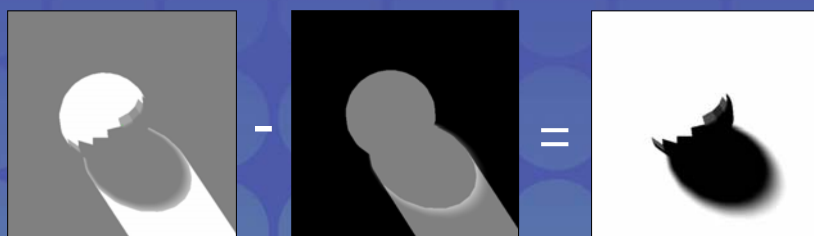
5 bits for overflow means that we can have <32 overlapping objects!



The last step shows how the RGBA is combined with a dot product to form the hires value that we need.

V-buffer (4)

- Clear V-buffer: 1.0 → additive, 0.0 → subtractive == fully lit
- Then, rasterize standard SV polys into V-buffer
  - +1.0 for frontfacing → subtractive buffer
  - +1.0 for backfacing → addritive buffer
- Rasterize penumbra wedges

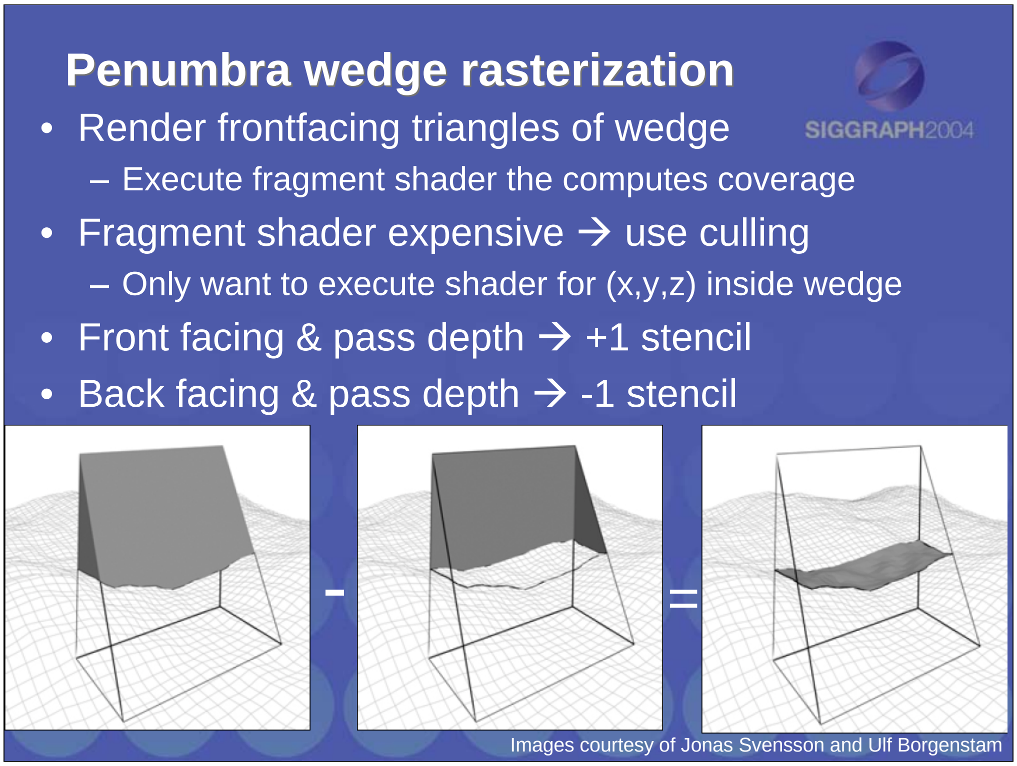


"additive buffer" "subtractive buffer" "soft shadow mask"

Images courtesy of Jonas Svensson and Ulf Borgenstam

Assume ZFAIL!

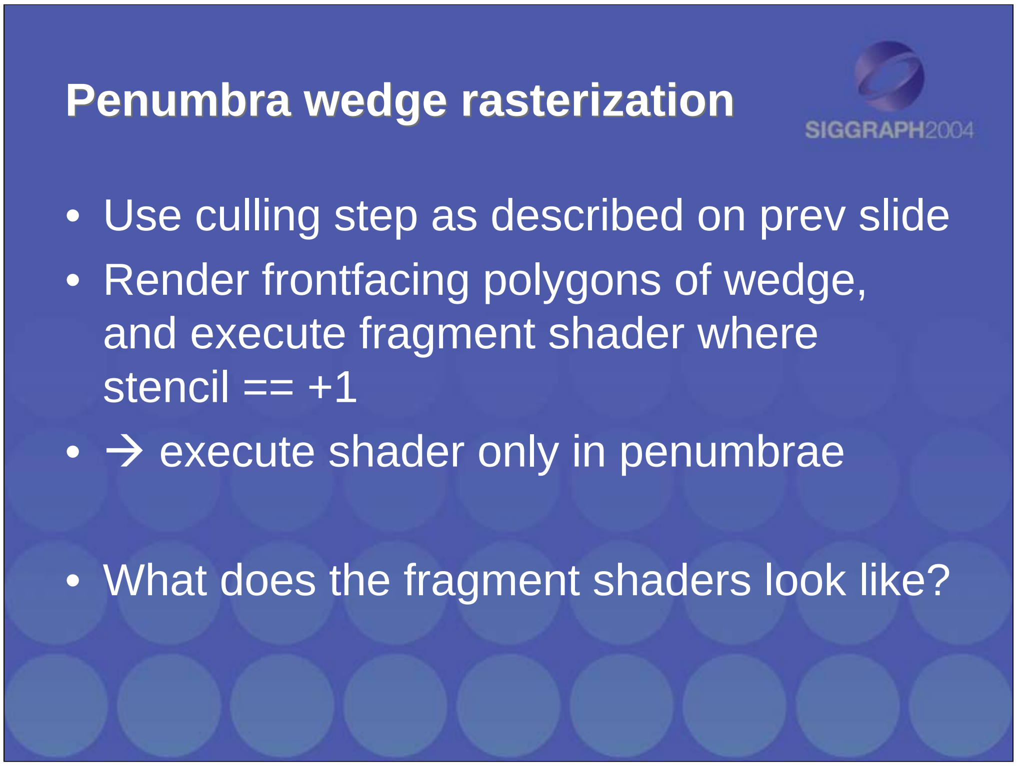
The result: ADD – SUB = FINAL SHADOW MASK



INSIGHT: use standard shadow volue algorithm on the wedge!

The images shows the ZPASS version.

Reduces artifacts!

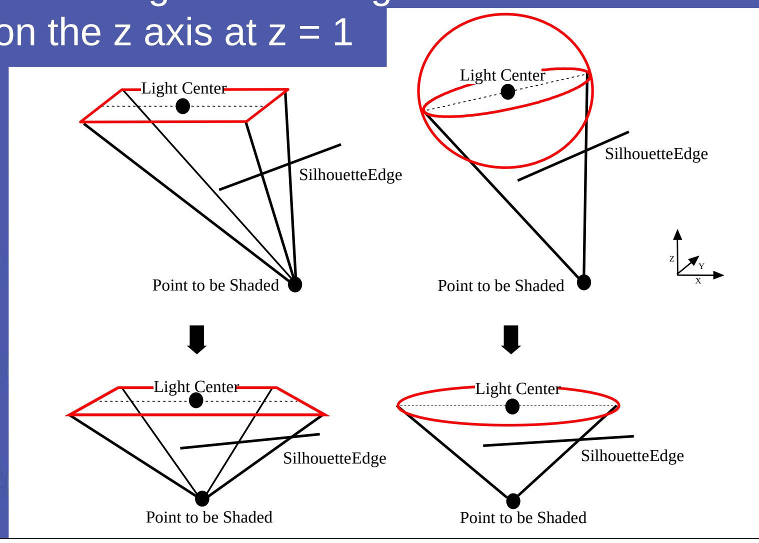


The culling helps performance quite a bit as well as avoiding artifacts that would otherwise come from testing whether a point is inside all wedge planes.

Fragment shaders: Transform the Silhouette Edge to Projection Space

SIGGRAPH2004

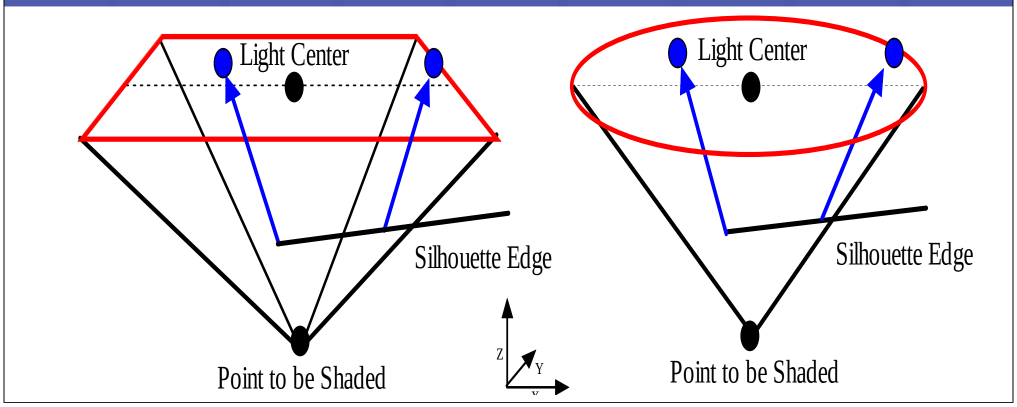
- Projection space is defined as the point to be shaded at the origin and the light center located on the z axis at z = 1



Projection is a SHEAR and a SCALE.

Clip edge and do perspective divide

- The square light shader clips edge to planes in homogenous space.
- The spherical light shader solves the equation of an intersection with a cone and line in homogenous space.
- The perspective divide is simply x/z and y/z.

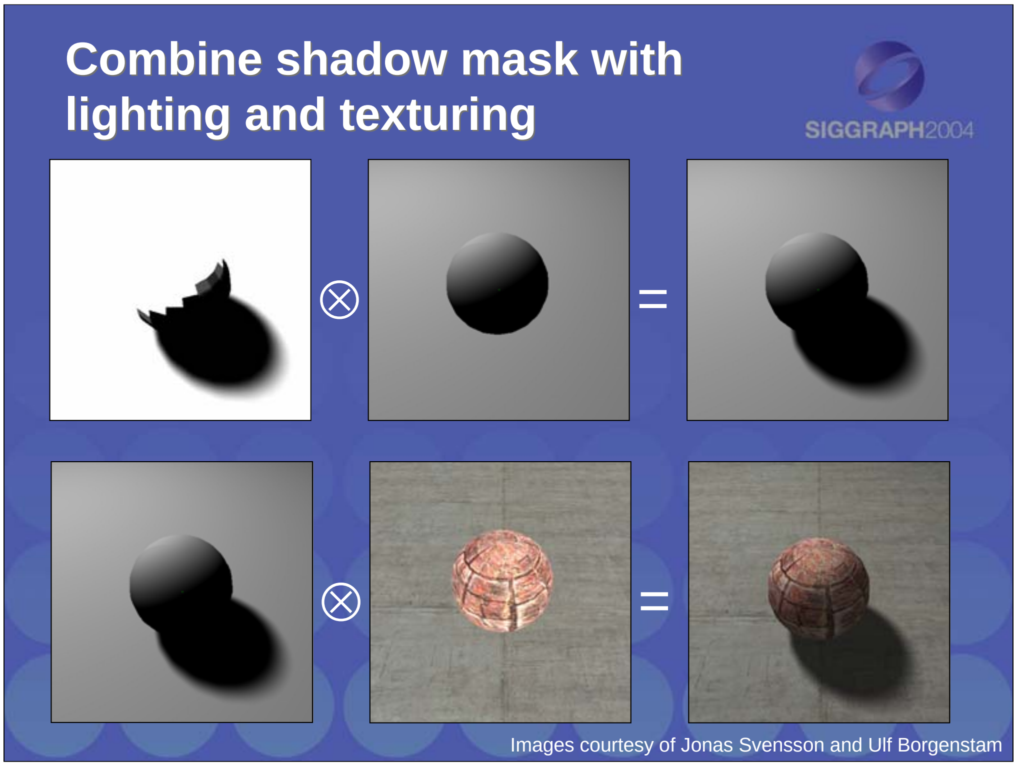


The shader programs become shorter when we performed clipping first, and then projection.

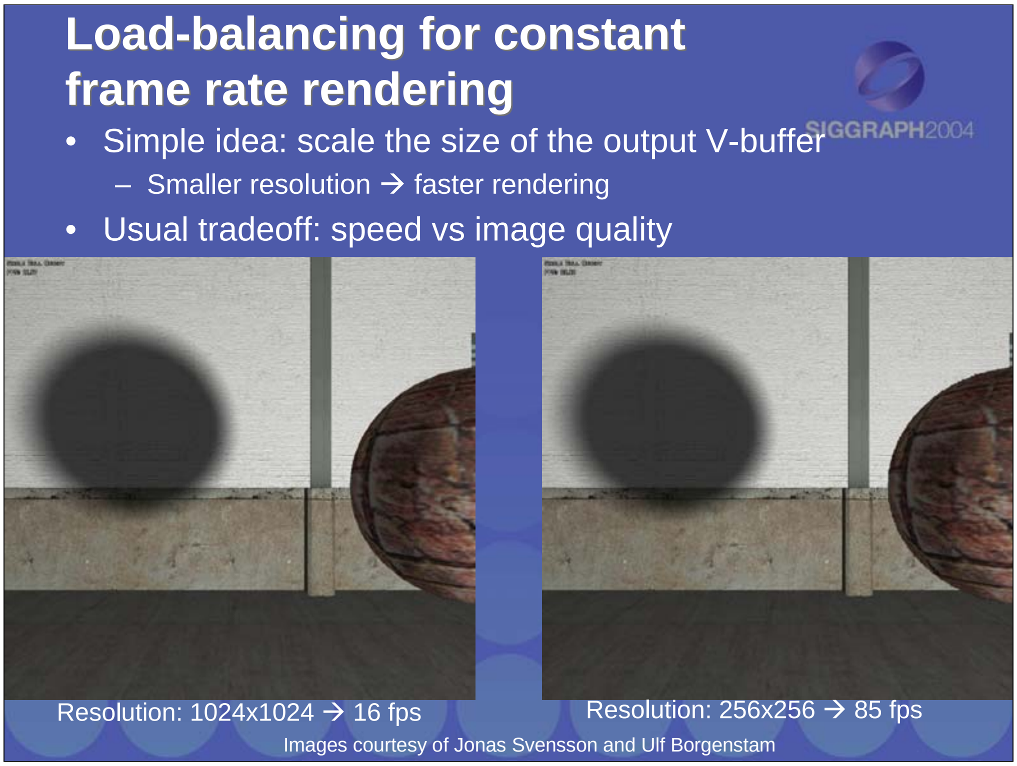
When edge has been clipped… compute coverage of edge

SIGGRAPH2004

- For rectangular lights:
  - – Use 4D coverage texture or
  - – Compute it analytically + using 2D textures (for constant lights)
    - Better accuracy!
- For spherical lights:
  - – Compute analytically + using 2D textures
- We’re done.



Or more EFFICIENTLY: render an image with diffuse lighting plus texturing, and then modulate with soft shadow mask.

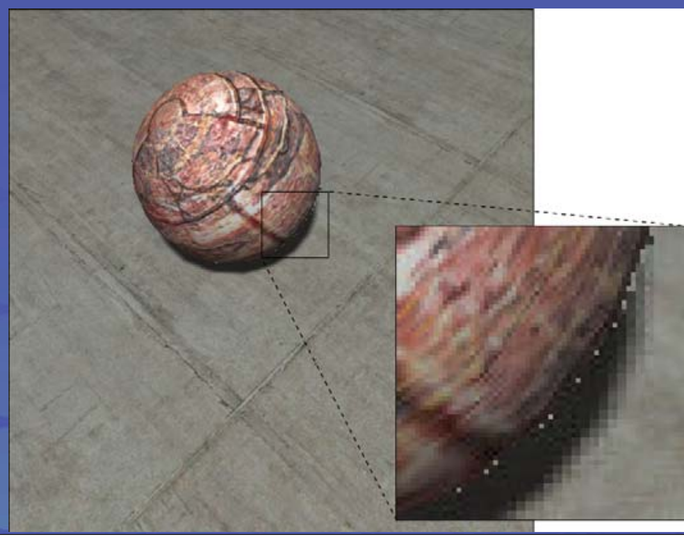


Always nice to be able to get higher frame rates. This is one way to do it.

Load balancing... cont'd

SIGGRAPH2004

- Use a simple reactive algorithm
- Will get rendering errors?

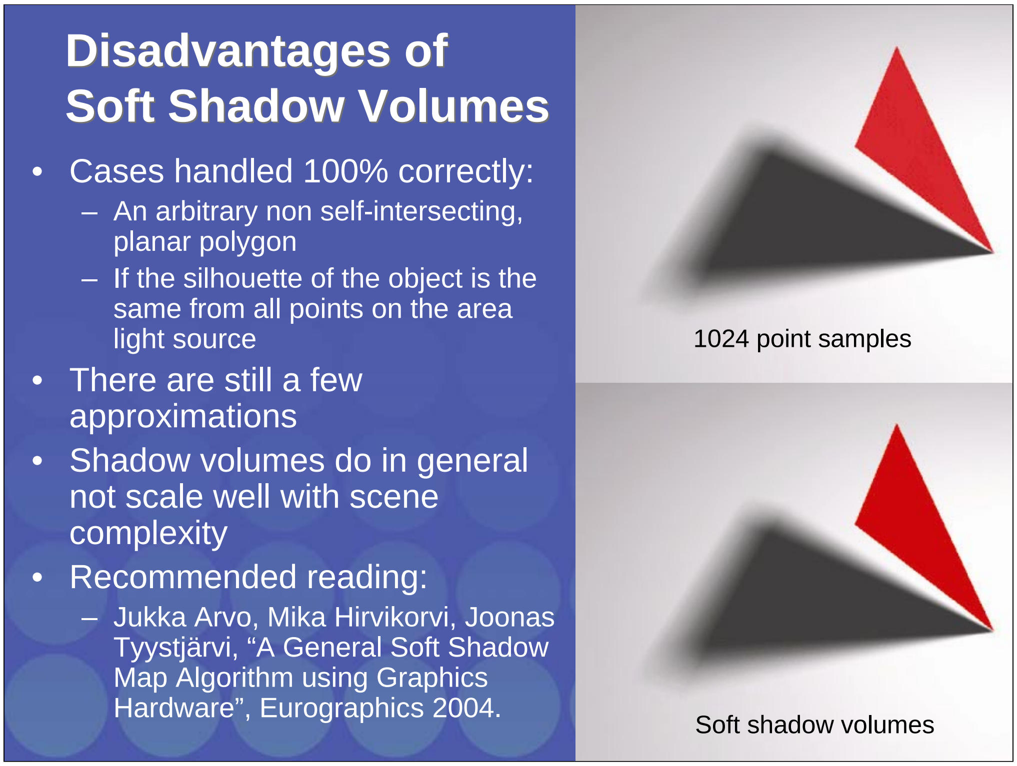


Images courtesy of Jonas Svensson and Ulf Borgenstam

Is it worth it?

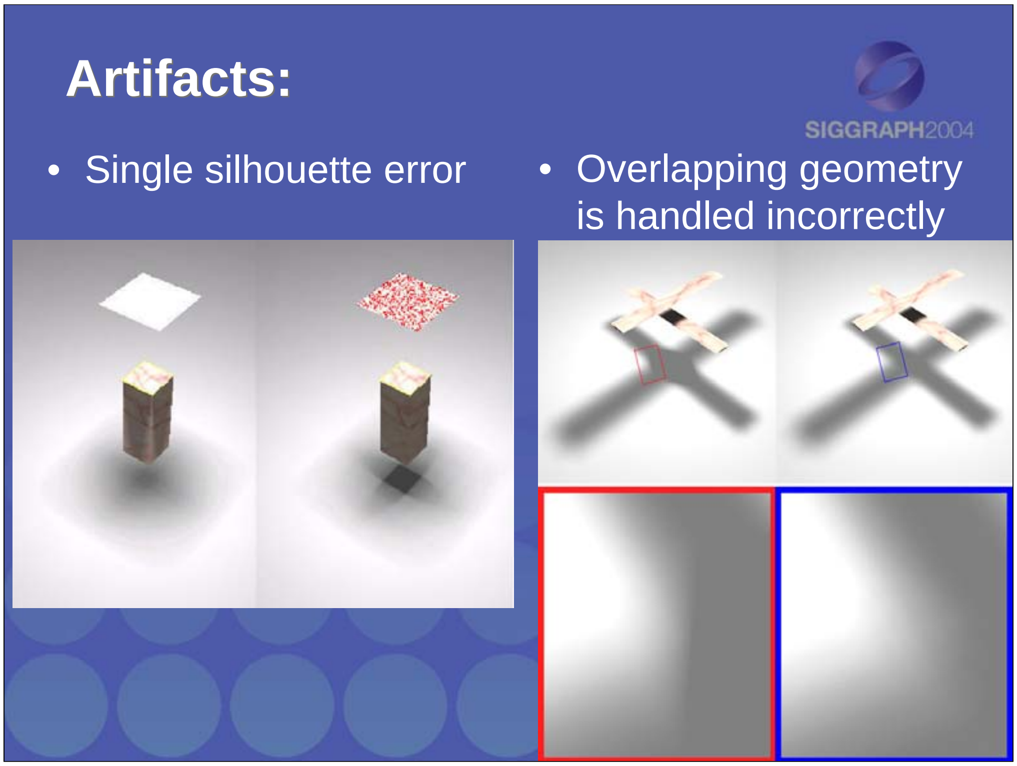
Will get flickering...

Can use bilinear filtering, but that costs...

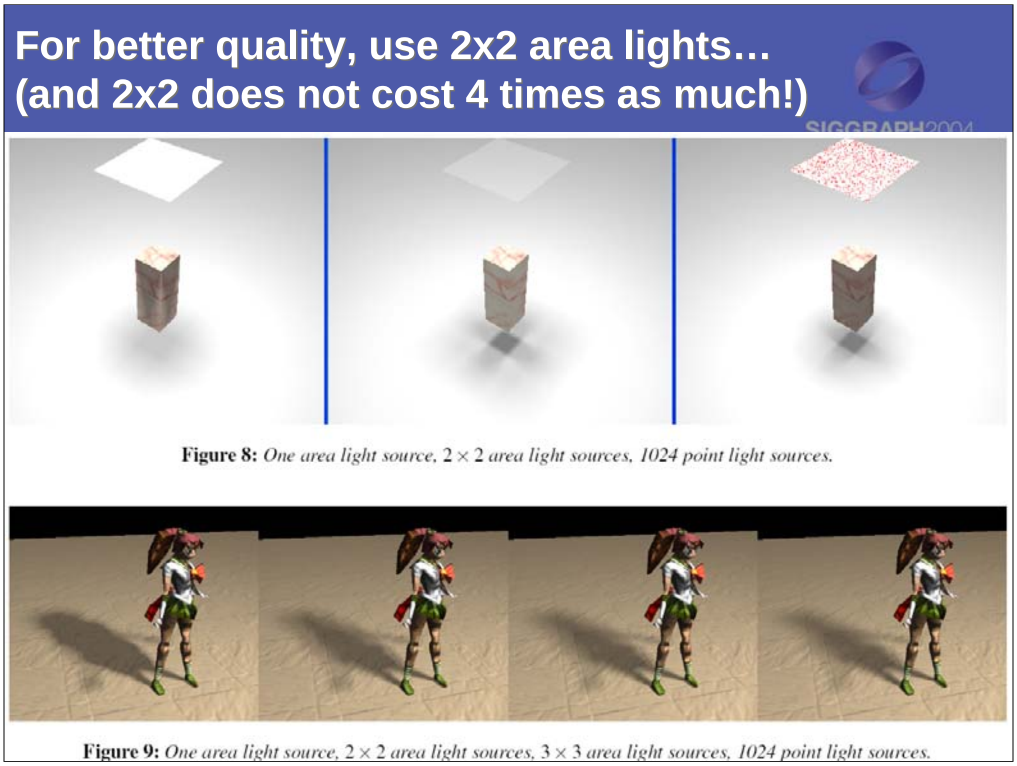


One further disadvantage is due to the single silhouette approximation: when the object is simple, the silhouette will change abrubptly as will the shadow...

Example of object: cube

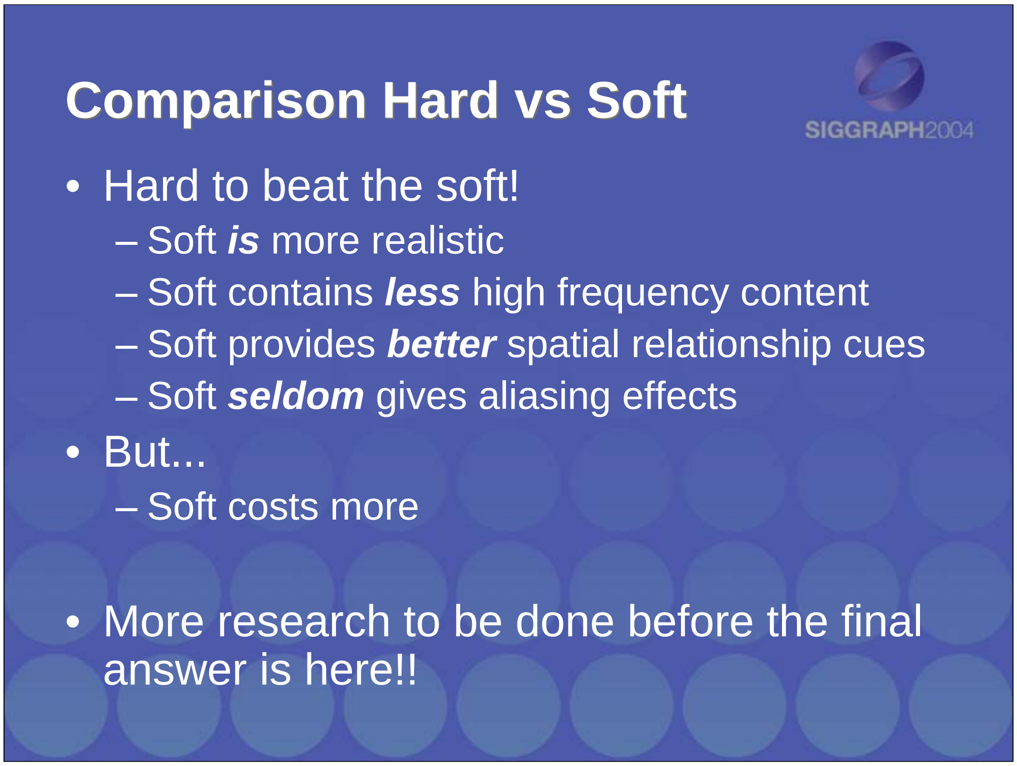


Single silhouette: can give a popping effect (e.g., for a cube with a moving light source → gives sudden and large changes of the silhouette).



In theory, we believe that 2x2 would cost about twice as much using ideal hardware...

Not sure what happens using real hardware.



When the shadow border contain only high frequency content (which is the case for hard shadows), the shadow boundary can be misinterpreted for a geometrical feature.

The battle goes on between shadow mapping and shadow volumes, both for hard and soft shadows. I do not favor either of these – we'll see in a few years.

Time for real-time demo...

- Frame work
  - Written in OpenGL using GL_ARB_fragment_program etc.
    - There is also a smaller DirectX demo coded by Michael Dougherty & Michael Mounier, XBOX Advanced group
  - Open and free source code
    - http://www.cs.lth.se/~tam/shadows/
  - Need graphics hardware:
    - ATI Radeon 9700 and up...
    - Any NVIDIA GeForce FX
  - Coded by: Jonas Svensson and Ulf Borgenstam

1024 hard shadow samples

Our algorithm

- Thanks for listening...
- This is ***not*** only my work – several persons contributed:
  – Ulf Assarsson, PhD on soft shadows
  – Michael Dougherty and Michael Mounier, DirectX implementation and optimization
  – Jonas Svensson & Ulf Borgenstam, OpenGL implementation